Capítulo 3

DIVIDE Y VENCERÁS

3.1 INTRODUCCIÓN

El término Divide y Vencerás en su acepción más amplia es algo más que una técnica de diseño de algoritmos. De hecho, suele ser considerada una filosofía general para resolver problemas y de aquí que su nombre no sólo forme parte del vocabulario informático, sino que también se utiliza en muchos otros ámbitos.

En nuestro contexto, Divide y Vencerás es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de subproblemas del mismo tipo, pero de menor tamaño. Si los subproblemas son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar subproblemas lo suficientemente pequeños para ser solucionados directamente. Ello naturalmente sugiere el uso de la recursión en las implementaciones de estos algoritmos.

La resolución de un problema mediante esta técnica consta fundamentalmente de los siguientes pasos:

1. En primer lugar ha de plantearse el problema de forma que pueda ser descompuesto en k subproblemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la entrada es n, hemos de conseguir dividir el problema en k subproblemas (donde $1 \le k \le n$), cada uno con una entrada de tamaño n_k y donde $0 \le n_k < n$. A esta tarea se le conoce como *división*.
2. En segundo lugar han de resolverse independientemente todos los subproblemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos *base*.
3. Por último, *combinar* las soluciones obtenidas en el paso anterior para construir la solución del problema original.

El funcionamiento de los algoritmos que siguen la técnica de Divide y Vencerás descrita anteriormente se refleja en el esquema general que presentamos a continuación:

```
PROCEDURE DyV(x:TipoProblema):TipoSolucion;
   VAR i,k,:CARDINAL;
       s:TipoSolucion;
       subproblemas: ARRAY OF TipoProblema;
       subsoluciones:ARRAY OF TipoSolucion;
BEGIN
   IF EsCasobase(x) THEN
      s:=ResuelveCasoBase(x)
   ELSE
      k:=Divide(x,subproblemas);
      FOR i:=1 TO k DO
         subsoluciones[i]:=DyV(subproblemas[i])
      END;
      s:=Combina(subsoluciones)
   END;
   RETURN s
END DyV;
```

Hemos de hacer unas apreciaciones en este esquema sobre el procedimiento *Divide*, sobre el número *k* que representa el número de subproblemas, y sobre el tamaño de los subproblemas, ya que de todo ello va a depender la eficiencia del algoritmo resultante.

En primer lugar, el número *k* debe ser pequeño e independiente de una entrada determinada. En el caso particular de los algoritmos Divide y Vencerás que contienen sólo una llamada recursiva, es decir $k = 1$, hablaremos de algoritmos de *simplificación*. Tal es el caso del algoritmo recursivo que resuelve el cálculo del factorial de un número, que sencillamente reduce el problema a otro subproblema del mismo tipo de tamaño más pequeño. También son algoritmos de simplificación el de búsqueda binaria en un vector o el que resuelve el problema del *k*-ésimo elemento.

La ventaja de los algoritmos de simplificación es que consiguen reducir el tamaño del problema en cada paso, por lo que sus tiempos de ejecución suelen ser muy buenos (normalmente de orden logarítmico o lineal). Además pueden admitir una mejora adicional, puesto que en ellos suele poder eliminarse fácilmente la recursión mediante el uso de un bucle iterativo, lo que conlleva menores tiempos de ejecución y menor complejidad espacial al no utilizar la pila de recursión, aunque por contra, también en detrimento de la legibilidad del código resultante.

Por el hecho de usar un diseño recursivo, los algoritmos diseñados mediante la técnica de Divide y Vencerás van a heredar las ventajas e inconvenientes que la recursión plantea:

a) Por un lado el diseño que se obtiene suele ser simple, claro, robusto y elegante, lo que da lugar a una mayor legibilidad y facilidad de depuración y mantenimiento del código obtenido.
b) Sin embargo, los diseños recursivos conllevan normalmente un mayor tiempo de ejecución que los iterativos, además de la complejidad espacial que puede representar el uso de la pila de recursión.

Desde un punto de vista de la eficiencia de los algoritmos Divide y Vencerás, es muy importante conseguir que los subproblemas sean independientes, es decir, que no exista solapamiento entre ellos. De lo contrario el tiempo de ejecución de estos algoritmos será exponencial. Como ejemplo pensemos en el cálculo de la sucesión de Fibonacci, el cual, a pesar de ajustarse al esquema general y de tener sólo dos llamadas recursivas, tan sólo se puede considerar un algoritmo recursivo pero no clasificarlo como diseño Divide y Vencerás. Esta técnica está concebida para resolver problemas de manera eficiente y evidentemente este algoritmo, con tiempo de ejecución exponencial, no lo es.

En cuanto a la eficiencia hay que tener en también en consideración un factor importante durante el diseño del algoritmo: el número de subproblemas y su tamaño, pues esto influye de forma notable en la complejidad del algoritmo resultante. Veámoslo más detenidamente.

En definitiva, el diseño Divide y Vencerás produce algoritmos recursivos cuyo tiempo de ejecución (según vimos en el primer capítulo) se puede expresar mediante una ecuación en recurrencia del tipo:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \le n < b \\ aT(n/b) + cn^k & \text{si } n \ge b \end{cases}$$

donde a, c y k son números reales, n y b son números naturales, y donde $a>0$, $c>0$, $k\geq 0$ y $b>1$. El valor de a representa el número de subproblemas, n/b es el tamaño de cada uno de ellos, y la expresión cn^k representa el coste de descomponer el problema inicial en los a subproblemas y el de combinar las soluciones para producir la solución del problema original, o bien el de resolver un problema elemental. La solución a esta ecuación, tal y como vimos en el problema 1.4 del primer capítulo, puede alcanzar distintas complejidades. Recordemos que el orden de complejidad de la solución a esta ecuación es:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Las diferencias surgen de los distintos valores que pueden tomar a y b, que en definitiva determinan el número de subproblemas y su tamaño. Lo importante es observar que en todos los casos la complejidad es de orden polinómico o polilogarítmico pero nunca exponencial, frente a los algoritmos recursivos que pueden alcanzar esta complejidad en muchos casos (véase el problema 1.3). Esto se debe normalmente a la repetición de los cálculos que se produce al existir solapamiento en los subproblemas en los que se descompone el problema original.

Para aquellos problemas en los que la solución haya de construirse a partir de las soluciones de subproblemas entre los que se produzca necesariamente solapamiento existe otra técnica de diseño más apropiada, y que permite eliminar el problema de la complejidad exponencial debida a la repetición de cálculos. Estamos hablando de la Programación Dinámica, que discutiremos en el capítulo cinco.

Otra consideración importante a la hora de diseñar algoritmos Divide y Vencerás es el reparto de la carga entre los subproblemas, puesto que es importante que la división en subproblemas se haga de la forma más equilibrada posible. En caso contrario nos podemos encontrar con "anomalías de funcionamiento" como le ocurre al algoritmo de ordenación Quicksort. Éste es un representante claro de los algoritmos Divide y Vencerás, y su caso peor aparece cuando existe un desequilibrio total en los subproblemas al descomponer el vector original en dos subvectores de tamaño 0 y n–1. Como vimos en el capítulo anterior, en este caso su orden es $O(n^2)$, frente a la buena complejidad, $O(n\log n)$, que consigue cuando descompone el vector en dos subvectores de igual tamaño.

También es interesante tener presente la dificultad y es esfuerzo requerido en cada una de estas fases va a depender del planteamiento del algoritmo concreto. Por ejemplo, los métodos de ordenación por Mezcla y Quicksort son dos representantes claros de esta técnica pues ambos están diseñados siguiendo el esquema presentado: dividir y combinar.

En lo que sigue del capítulo vamos a desarrollar una serie de ejemplos que ilustran esta técnica de diseño. Existe una serie de algoritmos considerados como representantes clásicos de este diseño, muy especialmente los de ordenación por Mezcla y Quicksort, que no incluimos en este capítulo por haber sido estudiados anteriormente. Sencillamente señalar la diferencia de esfuerzo que realizan en sus fases de división y combinación. La división de Quicksort es costosa, pero una vez ordenados los dos subvectores la combinación es inmediata. Sin embargo, la división que realiza el método de ordenación por Mezcla consiste simplemente en considerar la mitad de los elementos, mientras que su proceso de combinación es el que lleva asociado todo el esfuerzo.

Por último, y antes de comenzar con los ejemplos escogidos, sólo indicar que en muchos de los problemas aquí presentados haremos uso de vectores unidimensionales cuyo tipo viene dado por:

```
CONST n =...; (* numero maximo de elementos del vector *)
TYPE vector = ARRAY [1..n] OF INTEGER;
```

3.2 BÚSQUEDA BINARIA

El algoritmo de búsqueda binaria es un ejemplo claro de la técnica Divide y Vencerás. El problema de partida es decidir si existe un elemento dado x en un vector de enteros ordenado. El hecho de que esté ordenado va a permitir utilizar esta técnica, pues podemos plantear un algoritmo con la siguiente estrategia: compárese el elemento dado x con el que ocupa la posición central del vector. En caso de que coincida con él, hemos solucionado el problema. Pero si son distintos, pueden darse dos situaciones: que x sea mayor que el elemento en posición central, o que sea menor. En cualquiera de los dos casos podemos descartar una de las dos mitades del vector, puesto que si x es mayor que el elemento en posición central, también será mayor que todos los elementos en posiciones anteriores, y al revés. Ahora se procede de forma recursiva sobre la mitad que no hemos descartado.

En este ejemplo la división del problema es fácil, puesto que en cada paso se divide el vector en dos mitades tomando como referencia su posición central. El problema queda reducido a uno de menor tamaño y por ello hablamos de

"simplificación". Por supuesto, aquí no es necesario un proceso de combinación de resultados.

Su caso base se produce cuando el vector tiene sólo un elemento. En esta situación la solución del problema se basa en comparar dicho elemento con *x*. Como el tamaño de la entrada (en este caso el número de elementos del vector a tratar) se va dividiendo en cada paso por dos, tenemos asegurada la convergencia al caso base.

La función que implementa tal algoritmo ya ha sido expuesta en el primer capítulo como uno de los ejemplos de cálculo de complejidades (problema 1.16), con lo cual no reincidiremos más en ella.

3.3 BÚSQUEDA BINARIA NO CENTRADA

Una de las cuestiones a considerar cuando se diseña un algoritmo mediante la técnica de Divide y Vencerás es la partición y el reparto equilibrado de los subproblemas. Más concretamente, en el problema de la búsqueda binaria nos podemos plantear la siguiente cuestión: supongamos que en vez de dividir el vector de elementos en dos mitades del mismo tamaño, las dividimos en dos partes de tamaños 1/3 y 2/3. ¿Conseguiremos de esta forma un algoritmo mejor que el original?

Solución (☺)

Tal algoritmo puede ser implementado como sigue:

```
PROCEDURE BuscBin2(Var a:vector;
          prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
   VAR tercio:CARDINAL; (* posicion del elemento n/3 *)
BEGIN
   IF (prim>=ult) THEN RETURN a[ult]=x
   ELSE
      tercio:=prim+((ult-prim+1)DIV 3);
      IF x=a[tercio] THEN RETURN TRUE
      ELSIF (x<a[tercio]) THEN RETURN BuscBin2(a,prim,tercio,x)
      ELSE RETURN BuscBin2(a,tercio+1,ult,x)
      END
   END
END BuscBin2;
```

El cálculo del número de operaciones elementales que se realiza en el peor caso de una invocación a esta función puede hacerse de manera análoga a como se hizo en el problema 1.16 cuando se estudió la búsqueda binaria. En este caso obtenemos la ecuación en recurrencia $T(n) = 11 + T(2n/3)$, con la condición inicial $T(1) = 4$. Para resolverla, haciendo el cambio $t_k = T((3/2)^k)$ obtenemos

$$t_k - t_{k-1} = 11,$$

ecuación no homogénea con ecuación característica $(x–1)^2 = 0$. Por tanto,

$$t_k = c_1k + c_2$$

y, deshaciendo el cambio:

$$T(n) = c_1\log_{3/2}n + c_2.$$

Para calcular las constantes, nos basaremos en la condición inicial $T(1) = 4$, junto con el valor de $T(2)$, que puede ser calculado apoyándonos en la expresión de la ecuación en recurrencia, es decir, $T(2) = 11 + 4 = 15$. De esta forma obtenemos que:

$$T(n) = 11\log_{3/2}n + 4 \in \Theta(\log n).$$

A pesar de ser del mismo orden de complejidad que la búsqueda binaria clásica, como $3/2 < 2$ se tiene que $\log_{3/2}n > \log n$, es decir, este algoritmo es más lento en el caso peor que el algoritmo original presentado en el problema 1.16 del primer capítulo.

Este hecho puede ser generalizado fácilmente para demostrar que, dividiendo el vector en dos partes de tamaños k y $n–k$, el tiempo de ejecución del algoritmo resultante en el peor caso es:

$$T_k(n) = 11\log_{n/\max\{k,n-k\}}n + 4 \in \Theta(\log n).$$

Ahora bien, para $1 \leq k < n$ sabemos que la función $n/\max\{k,n–k\}$ se mantiene por debajo de 2, y sólo alcanza este valor para $k = n/2$, por lo que

$$\log_{n/\max\{k,n-k\}}n \geq \log n$$

para todo k entre 1 y n. Esto nos indica que la mejor forma de partir el vector para realizar la búsqueda binaria es por la mitad, es decir, tratando de equilibrar los subproblemas en los que realizamos la división tal como comentábamos en la introducción de este capítulo.

3.4 BÚSQUEDA TERNARIA

Podemos plantearnos también diseñar un algoritmo de búsqueda "ternaria", que primero compara con el elemento en posición $n/3$ del vector, si éste es menor que el elemento x a buscar entonces compara con el elemento en posición $2n/3$, y si no coincide con x busca recursivamente en el correspondiente subvector de tamaño 1/3 del original. ¿Conseguimos así un algoritmo mejor que el de búsqueda binaria?

Solución (☺)

Podemos implementar el algoritmo pedido, también de simplificación, de una forma similar a la anterior. La única diferencia es la interpretación de la variable *nterc*, que no indica una posición dentro del vector (como le ocurría a la variable *tercio* del ejercicio previo), sino el número de elementos a tratar ($n/3$). Es por eso por lo que se lo sumamos y restamos a los valores de *prim* y *ult* para obtener las posiciones adecuadas. El algoritmo resultante es:

```
PROCEDURE BuscBin3(VAR a:vector;prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
   VAR nterc:CARDINAL; (* 1/3 del numero de elementos *)
BEGIN
   IF (prim>=ult) THEN RETURN a[ult]=x END;         (* 1 *)
   nterc:=(ult-prim+1)DIV 3;                        (* 2 *)
   IF x=a[prim+nterc] THEN RETURN TRUE        (* 3 *)
   ELSIF x<a[prim+nterc] THEN                       (* 4 *)
      RETURN BuscBin3(a,prim,prim+nterc-1,x)   (* 5 *)
   ELSIF x=a[ult-nterc] THEN RETURN TRUE            (* 6 *)
   ELSIF x<a[ult-nterc] THEN                        (* 7 *)
      RETURN BuscBin3(a,prim+nterc+1,ult-nterc-1,x) (* 8 *)
   ELSE                                             (* 9 *)
      RETURN BuscBin3(a,ult-nterc+1,ult,x)          (* 10 *)
   END                                              (* 11 *)
   END                                              (* 12 *)
END BuscBin3;
```

Para estudiar su complejidad calcularemos el número de operaciones elementales que se realizan:

– En la línea (1) se ejecutan la comparación del *IF* (1 OE), y un acceso a un vector (1 OE), una comparación (1 OE) y un *RETURN* (1 OE) si la condición es verdadera.
– En la línea (2) se realizan 4 OE (resta, suma, división y asignación).
– En la línea (3) hay una suma (1 OE), un acceso a un vector (1 OE) y una comparación (1 OE), más 1 OE si la condición del *IF* es verdadera.
– En la línea (4) se realiza una suma (1 OE), un acceso a un vector (1 OE) y una comparación (1 OE).
– En la línea (5) se efectúan 2 operaciones aritméticas (2 OE), una llamada a la función *BuscBin3* (lo que supone 1 OE), más lo que tarde en ejecutarse la función con un tercio de los elementos y un *RETURN* (1 OE).
– Las líneas (6) y (7) suponen las mismas OE que las líneas (3) y (4).
– Por último, las líneas (8) y (10) efectúan $6+T(n/3)$ y $4+T(n/3)$ cada una: 4 y 2 operaciones aritméticas respectivamente, una llamada a la función *BuscBin3* (lo que supone 1 OE), más lo que tarde en ejecutarse la función con un tercio de los elementos y un *RETURN* (1 OE).

Por tanto, en el peor caso obtenemos la ecuación $T(n) = 23 + T(n/3)$, con la condición inicial $T(1) = 4$. Para resolverla, haciendo el cambio $t_k=T(3^k)$ obtenemos

$$t_k - t_{k-1} = 23,$$

ecuación no homogénea cuya ecuación característica es $(x-1)^2 = 0$. Por tanto, $t_k = c_1k + c_2$ y, deshaciendo el cambio,

$$T(n) = c_1\log_3 n + c_2.$$

Para calcular las constantes, nos basaremos en la condición inicial $T(1) = 4$, junto con el valor de $T(3)$, que puede ser calculado apoyándonos en la expresión de la ecuación en recurrencia: $T(3) = 23 + 4 = 27$. De esta forma obtenemos:

$$T(n) = 23\log_3 n + 4 \in \Theta(\log n).$$

Como $23\log_3 n = 23\log n/\log 3 = 14.51\log n > 11\log n$, el tiempo de ejecución de la búsqueda ternaria es mayor al de la búsqueda binaria, por lo que no consigue ninguna mejora con este algoritmo.

3.5 MULTIPLICACIÓN DE ENTEROS

Sean u y v dos números naturales de n bits donde, por simplicidad, n es una potencia de 2. El algoritmo tradicional para multiplicarlos es de complejidad $O(n^2)$. Ahora bien, un algoritmo basado en la técnica de Divide y Vencerás expuesto en [AHO87] divide los números en dos partes

$$u = a2^{n/2} + b$$
$$v = c2^{n/2} + d$$

siendo a, b, c y d números naturales de $n/2$ bits, y calcula su producto como sigue:

$$uv = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd$$

Las multiplicaciones ac, ad, bc y bd se realizan usando este algoritmo recursivamente. En primer lugar nos gustaría estudiar la complejidad de este algoritmo para ver si ofrece alguna mejora frente al tradicional.

Por otro lado podríamos pensar en otro algoritmo Divide y Vencerás en el que la expresión $ad+bc$ la sustituimos por la expresión equivalente $(a-b)(d-c)+ac+bd$. Nos cuestionamos si se consigue así un algoritmo mejor que el anterior.

Solución (☺)

Necesitamos en primer lugar determinar su caso base, que en este caso ocurre para $n = 1$, es decir, cuando los dos números son de 1 bit, en cuyo caso uv vale 1 si $u = v = 1$, o bien 0 en otro caso.

Para compararlo con otros algoritmos es necesario determinar su tiempo de ejecución y complejidad. Para ello hemos de observar que para calcular una multiplicación de dos números de n bits –$T(n)$– es necesario realizar cuatro multiplicaciones de $n/2$ bits (las de ac, ad, bc y bd), dos desplazamientos (las multiplicaciones por 2^n y $2^{n/2}$) y tres sumas de números de a lo más $2n$ bits (en el peor caso, ése es el tamaño del mayor de los tres números, pues ac puede alcanzar n bits). Como las sumas y los desplazamientos son operaciones de orden n, el orden de complejidad del algoritmo viene dada por la expresión:

$$T(n) = 4T(n/2) + An,$$

siendo A una constante. Además, podemos tomar $T(1) = 1$. Para resolver la ecuación hacemos el cambio $t_k = T(2^k)$, obteniendo

$$t_k = 4t_{k-1} + A2^k,$$

ecuación en recurrencia no homogénea con ecuación característica $(x–4)(x–2) = 0$. Aplicando los métodos utilizados en el capítulo 1, la expresión de t_k viene dada por

$$t_k = c_1 4^k + c_2 2^k,$$

y de aquí, deshaciendo el cambio $n = 2^k$ (o lo que es igual, $k = \log n$), obtenemos

$$T(n) = c_1 4^{\log n} + c_2 2^{\log n} = c_1 n^2 + c_2 n \in O(n^2).$$

Por tanto, este método no mejora el tradicional, también de orden n^2. En cuanto a la modificación sugerida, expresando $ad+bc$ como $(a–b)(d–c)+ac+bd$ obtenemos la siguiente expresión para uv:

$$uv = ac2^n + ((a–b)(d–c)+ac+bd)2^{n/2} + bd.$$

Aunque aparentemente es más complicada, su cálculo precisa tan sólo de tres multiplicaciones de números de $n/2$ bits (ac, bd y $(a–b)(d–c)$) dos desplazamientos de números de n y $n/2$ bits, y seis sumas de números de a lo más $2n$ bits. Tanto las sumas como los desplazamientos son de orden n, y en consecuencia el tiempo de ejecución del algoritmo viene dado por la expresión

$$T(n) = 3T(n/2) + Bn,$$

siendo B una constante. Nuestro caso base sigue siendo el mismo, por lo que podemos volver a tomar $T(1) = 1$. Para resolver la ecuación hacemos el cambio $t_k = T(2^k)$ y obtenemos:

$$t_k = 3t_{k-1} + B2^k,$$

ecuación en recurrencia no homogénea con ecuación característica $(x–3)(x–2) = 0$. Aplicando de nuevo los métodos utilizados en el capítulo 1, la expresión de t_k viene dada por

$$t_k = c_1 3^k + c_2 2^k,$$

y de aquí, deshaciendo el cambio $n = 2^k$ (o lo que es igual, $k = \log n$), obtenemos:

$$T(n) = c_1 3^{\log n} + c_2 2^{\log n} = c_1 n^{\log 3} + c_2 n \in O(n^{1.59}).$$

Por tanto, este método es de un orden de complejidad menor que el tradicional. ¿Por qué no se enseña entonces en las escuelas y se usa normalmente? Existen fundamentalmente dos razones para ello, una de las cuales es que, aunque más eficiente, es mucho menos intuitivo que el método clásico. La segunda es que las constantes de proporcionalidad que se obtienen en esta caso hacen que el nuevo método sea más eficiente que el tradicional a partir de 500 bits (cf. [AHO87]), y los números que normalmente multiplicamos a mano son, afortunadamente, menores de ese tamaño.

3.6 PRODUCTO DE MATRICES CUADRADAS (1)

Supongamos que necesitamos calcular el producto de matrices cuadradas de orden n, donde n es una potencia de 3. Usando la técnica de Divide y Vencerás, el problema puede ser reducido a una multiplicación de matrices cuadradas de orden 3. El método tradicional para multiplicar estas matrices requiere 27 multiplicaciones. ¿Cuántas multiplicaciones hemos de ser capaces de realizar para multiplicar dos matrices cuadradas de orden 3 para obtener un tiempo total del algoritmo menor que $O(n^{2.81})$? De forma análoga podemos plantearnos la misma pregunta para el caso de matrices cuadradas de orden n, con n una potencia de 4.

Solución (☺)

Utilizando el método tradicional para multiplicar matrices cuadradas de orden tres necesitamos 27 multiplicaciones escalares. Por tanto, basándonos en él para multiplicar matrices cuadradas de orden $n = 3^k$ (multiplicando por bloques), obtenemos que el número de multiplicaciones escalares requerido para este caso (despreciando las adiciones) viene dado por la ecuación en recurrencia

$$T(3^k) = 27T(3^{k-1})$$

con la condición inicial $T(3) = 27$. Resolviendo esta ecuación homogénea, obtenemos que $T(n) = n^3$, resultado clásico ya conocido.

Sea ahora M el número pedido, que indica el número de multiplicaciones escalares necesario para multiplicar dos matrices cuadradas de orden 3. Entonces el número total de multiplicaciones necesario para multiplicar dos matrices cuadradas de orden $n = 3^k$ (multiplicando por bloques) vendrá dado por la ecuación:

$$T(3^k) = M \cdot T(3^{k-1})$$

con la condición inicial $T(3) = M$. Resolviendo esta ecuación homogénea,

$$T(n) = n^{\log_3 M}.$$

Para que la complejidad de $T(n)$ sea menor que $O(n^{2.81})$ se ha de cumplir que $\log_3 M < 2.81$. Por tanto, M ha de verificar que

$$M < 3^{2.81} \approx 22.$$

Es decir, necesitamos encontrar un método para multiplicar matrices de orden 3 con 21 o menos multiplicaciones escalares, en vez de las 27 usuales.

Pasemos ahora al caso de matrices cuadradas de orden $n = 4^k$, y sea N el número de multiplicaciones escalares necesario para multiplicar dos matrices cuadradas de orden 4. En este caso obtenemos la ecuación en recurrencia:

$$T(4^k) = N \cdot T(4^{k-1})$$

con la condición inicial $T(4) = N$. Resolviendo esta ecuación homogénea, obtenemos que

$$T(n) = n^{\log_4 N}.$$

Para que la complejidad de $T(n)$ sea menor que $O(n^{2.81})$ se ha de cumplir que $\log_4 N < 2.81$. Por tanto, N ha de verificar que

$$N < 4^{2.81} \approx 49.$$

Es decir, necesitamos encontrar un método para multiplicar matrices de orden 4 con 48 o menos multiplicaciones escalares, en vez de las 64 ($=4^3$) usuales.

Este tipo de problemas tiene su origen en el descubrimiento de Strassen (1968), que diseñó un método para multiplicar matrices cuadradas de orden 2 usando sólo siete multiplicaciones escalares, en vez de las ocho necesarias en el método clásico (despreciando las adiciones frente a las multiplicaciones). Así se consigue un algoritmo de multiplicación de matrices cuadradas nxn del orden de $n^{\log 7} = n^{2.81}$ en vez de los clásicos $n^{\log 8} = n^3$.

Dadas dos matrices cuadradas de orden 2, A y B, tal algoritmo se basa en obtener la matriz producto C mediante las siguientes fórmulas:

$$c_{11} = m_1 + m_2 - m_4 + m_6$$
$$c_{12} = m_4 + m_5$$
$$c_{21} = m_6 + m_7$$
$$c_{22} = m_2 - m_3 + m_5 - m_7$$

donde los valores de $m_1, m_2, ..., m_7$ vienen dados por:

$$m_1 = (a_{12} - a_{22})(b_{21} + b_{22})$$
$$m_2 = (a_{11} + a_{22})(b_{11} + b_{22})$$
$$m_3 = (a_{11} - a_{21})(b_{11} + b_{12})$$
$$m_4 = (a_{11} + a_{12})b_{22}$$
$$m_5 = a_{11}(b_{12} - b_{22})$$
$$m_6 = a_{22}(b_{21} - b_{11})$$
$$m_7 = (a_{21} + a_{22})b_{11}$$

Aunque el número de sumas se ha visto incrementado, el número de multiplicaciones escalares se ha reducido a siete. Utilizando este método para multiplicar por bloques dos matrices cuadradas de orden n, con n potencia de 2, conseguimos un algoritmo cuyo tiempo de ejecución viene dado por la ecuación en recurrencia $T(2^k) = 7T(2^{k-1})$, con la condición inicial $T(2) = 7$. Resolviendo esta ecuación homogénea, obtenemos

$$T(n) = n^{\log 7} \approx n^{2.81}.$$

Para intentar mejorar el algoritmo de Strassen, una primera idea es la de conseguir multiplicar matrices cuadradas de orden 2 con sólo seis multiplicaciones escalares, lo que llevaría a un método de orden $n^{\log 6} < n^{2.81}$. Sin embargo, Hopcroft y Kerr probaron en 1971 que esto es imposible.

Lo siguiente es pensar en matrices cuadradas de orden 3. Según acabamos de ver, si consiguiésemos un método para multiplicar dos matrices de orden 3 con 21 o menos multiplicaciones escalares conseguiríamos un método mejor que el de Strassen. Sin embargo, esto también se ha demostrado que es imposible.

¿Y para matrices de otros órdenes? En general, queremos encontrar un método para multiplicar matrices cuadradas de orden k utilizando menos de $k^{2.81}$

multiplicaciones escalares, en vez de las k^3 requeridas por el método clásico. El primer k que se descubrió fue $k = 70$, y se han llegado a conseguir métodos hasta de orden de $n^{2.376}$, al menos en teoría.

Sin embargo todos estos métodos no tienen ninguna utilidad práctica, debido a las constantes multiplicativas que poseen. Sólo el de Strassen es quizá el único útil, aún teniendo en cuenta que incluso él no muestra su bondad hasta valores de n muy grandes, pues en la práctica no podemos despreciar las adiciones. Incluso para tales matrices la mejora real obtenida es sólo de $n^{1.5}$ frente a $n^{1.41}$, lo que hace de este algoritmo una contribución más teórica que práctica por la dificultad en su codificación y mantenimiento.

3.7 PRODUCTO DE MATRICES CUADRADAS (2)

Sean $n = 2p$, $V = (v_1,v_2,...,v_n)$ y $W = (w_1,w_2,...,w_n)$. Para calcular el producto escalar de ambos vectores podemos usar la fórmula:

$$V\cdot W = \sum_{i=1}^{p}(v_{2i-1} + w_{2i})(v_{2i} + w_{2i-1}) - \sum_{i=1}^{p} v_{2i-1}v_{2i} - \sum_{i=1}^{p} w_{2i-1}w_{2i}$$

que requiere $3n/2$ multiplicaciones. ¿Podemos utilizar esta fórmula para la multiplicación de matrices cuadradas de orden n dando lugar a un método que requiera del orden de $n^3/2 + n^2$ multiplicaciones, en vez de las usuales n^3?

Solución (☹)

La multiplicación de dos matrices cuadradas de orden n puede realizarse utilizando el método clásico, que consiste en realizar n^2 multiplicaciones escalares de vectores: supongamos que queremos calcular el producto $A = B\cdot C$, siendo A, B y C tres matrices cuadradas de orden n.

Llamando B_i a los vectores fila de B (i=1,...,n), y C^j a los vectores columna de C (j=1,...,n),

$$B = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} \qquad C = \begin{pmatrix} C^1 & C^2 & \dots & C^n \end{pmatrix}$$

obtenemos que $A[i,j] = B_i\cdot C^j$ ($1\le i\le n$, $1\le j\le n$), es decir, cada elemento de A puede calcularse como la multiplicación escalar de dos vectores. Por tanto, son necesarias n^2 multiplicaciones de vectores (una para cada elemento de A).

El método usual de multiplicar escalarmente dos vectores V y W da lugar a n multiplicaciones de elementos, si es que se utiliza la fórmula:

$$V\cdot W = \sum_{i=1}^{n} v_i w_i .$$

Ahora bien, estudiando la fórmula propuesta para el caso en que n es par ($n = 2p$):

$$V{\cdot}W = \sum_{i=1}^{p}(v_{2i-1} + w_{2i})(v_{2i} + w_{2i-1}) - \sum_{i=1}^{p} v_{2i-1}v_{2i} - \sum_{i=1}^{p} w_{2i-1}w_{2i}$$

podemos observar que pueden reutilizarse muchos cálculos, puesto que los dos últimos sumandos de esta ecuación sólo dependen de los vectores V y W. Entonces el método pedido puede implementarse como sigue:

- Primero se calculan las sumas

  $$\sum_{i=1}^{p} v_{2i-1}v_{2i}$$

  para cada uno de los vectores fila de B y columna de C. Hay que realizar $2n$ de estas operaciones, y cada una requiere $n/2$ multiplicaciones, lo que implica n^2 multiplicaciones de elementos en esta fase.
- Después se calcula cada uno de los elementos de A como $A[i,j] = B_i{\cdot}C^j$ utilizando la fórmula anterior, pero en donde ya hemos calculado (en el paso previo) los dos últimos términos de los tres que componen la expresión. Por tanto, para cada elemento de A sólo es necesario realizar ahora $n/2$ multiplicaciones de elementos. Como hay que calcular n^2 elementos en total, realizaremos en esta fase $n^3/2$ multiplicaciones.

Sumando el número de multiplicaciones de ambas fases, hemos conseguido un método con el número de operaciones pedido.

3.8 MEDIANA DE DOS VECTORES

Sean X e Y dos vectores de tamaño n, ordenados de forma no decreciente. Necesitamos implementar un algoritmo para calcular la *mediana* de los $2n$ elementos que contienen X e Y. Recordemos que la mediana de un vector de k elementos es aquel elemento que ocupa la posición $(k+1)\div 2$ una vez el vector está ordenado de forma creciente. Dicho de otra forma, la mediada es aquel elemento que, una vez ordenado el vector, deja la mitad de los elementos a cada uno de sus lados. Como en nuestro caso $k = 2n$ (y por tanto par) buscamos el elemento en posición n de la unión ordenada de X e Y.

Solución (☺)

Para resolver el problema utilizaremos una idea basada en el método de búsqueda binaria. Comenzaremos estudiando el caso base, que ocurre cuando tenemos dos vectores de un elemento cada uno ($n = 1$). En este caso la mediana será el mínimo de ambos números, pues obedeciendo a la definición sería el elemento que ocupa la primera posición ($n = 1$) si ordenásemos ambos vectores.

Respecto al caso general, existe una forma de dividir el problema en subproblemas más pequeños. Sea Z el vector resultante de mezclar ordenadamente los vectores X e Y, y sea m_Z la mediana de Z. Apoyándonos en el hecho de que X e Y se encuentran ordenados, es fácil calcular sus medianas (son los elementos que ocupan las posiciones centrales de ambos vectores), y que llamaremos m_X y m_Y.

Ahora bien, si $m_X = m_Y$ entonces la mediana m_Z va a coincidir también con ellas, pues al mezclar ambos vectores las medianas se situarán en el centro del vector.

Si tenemos que $m_X < m_Y$, podemos afirmar que la mediana m_Z va a ser mayor que m_X pero menor que m_Y, y por tanto m_Z va a encontrarse en algún lugar de la segunda mitad del vector X o en algún lugar de la primera mitad de Y (por estar ambos vectores ordenados).

Análogamente, si $m_X > m_Y$ la mediana m_Z va a ser mayor que m_Y pero menor que m_X, y por tanto m_Z va a encontrarse en algún lugar de la primera mitad del vector X o en algún lugar de la segunda mitad de Y (por estar ambos vectores ordenados). Esta idea nos lleva a la siguiente versión de nuestro algoritmo:

```
PROCEDURE Mediana(VAR X,Y:vector;primX,ultX,primY,ultY:CARDINAL)
                                                        :INTEGER;
   VAR posX,posY:CARDINAL; nitems:CARDINAL;
BEGIN
   IF (primX>=ultX) AND (primY>=ultY) THEN (* caso base *)
      RETURN Min2(X[ultX],Y[ultY])
   END;
   nitems:=ultX-primX+1;
   IF nitems=2 THEN      (* 2 vectores de 2 elementos cada uno *)
      IF X[ultX]<Y[primY] THEN RETURN X[ultX]
      ELSIF Y[ultY]<X[primX] THEN RETURN Y[ultY]
      ELSE RETURN Max2(X[primX],Y[primY])
      END
   END;
   nitems:=(nitems-1) DIV 2;  (* caso general *)
   posX:=primX+nitems;
   posY:=primY+nitems;
   IF X[posX]=Y[posY] THEN RETURN X[posX]
   ELSIF X[posX]<Y[posY] THEN
      RETURN Mediana(X,Y,ultX-nitems,ultX,primY,primY+nitems)
   ELSE
      RETURN Mediana(X,Y,primX,primX+nitems,ultY-nitems,ultY)
   END;
END Mediana;
```

que calcula la solución pedida cuando lo invocamos como *Mediana*(X,Y,1,*n*,1,*n*). Es conveniente observar que uno de los invariantes del algoritmo es que el número de elementos de los subvectores X e Y coincide en cada uno de los pasos (es decir, se verifica que *ultX–primX+1=ultY–primY+1=n*) y que en cada invocación se reduce a la mitad, y por ello se trata de un algoritmo de simplificación. Este hecho de que el tamaño de los dos vectores es siempre igual en cada invocación es lo que nos permite garantizar que la media que se calcula recursivamente en los trozos coincide con la mediana buscada antes de realizar los descartes de elementos. En particular, basta con observar que los trozos descartados eliminan el mismo número de elementos.

En otro orden de cosas, las funciones *Min2* y *Max2* que utiliza este algoritmo son las que calculan respectivamente el mínimo y el máximo de dos números enteros.

Para el estudio de su complejidad, expresamos su tiempo de ejecución como

$$T(2n) = T(n) + A,$$

siendo A una constante. Entonces hacemos el cambio $t_k = T(2^k)$, y entonces

$$t_{k+1} = t_k + A,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-1)^2 = 0$. Aplicando los métodos utilizados en el capítulo 1, la expresión de t_k viene dada por

$$t_k = c_1 k + c_2,$$

y de aquí, deshaciendo el cambio $n = 2^k$ (o lo que es igual, $k = \log n$), obtenemos:

$$T(n) = c_1 \log n + c_2 \in O(\log n).$$

3.9 EL ELEMENTO EN SU POSICIÓN

Sea $a[1..n]$ un vector ordenado de enteros todos distintos. Nuestro problema es implementar un algoritmo de complejidad $O(\log n)$ en el peor caso capaz de encontrar un índice i tal que $1 \le i \le n$ y $a[i] = i$, suponiendo que tal índice exista.

Solución (☺)

Podemos implementar el algoritmo pedido apoyándonos en el hecho de que el vector está originalmente ordenado. Por tanto, podemos usar un método basado en la idea de la búsqueda binaria, en donde examinamos el elemento en mitad del vector (su mediana). Si $a[(n+1)\div 2] = (n+1)\div 2$, ésa es la posición pedida. Si $a[(n+1)\div 2]$ fuera mayor que $(n+1)\div 2$, la posición pedida ha de encontrarse antes de la mitad, y en caso contrario detrás de ella.

Esto da lugar al siguiente algoritmo:

```
PROCEDURE Localiza(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
   VAR i:CARDINAL;
BEGIN
   IF prim>ult THEN RETURN 0 END; (* no existe tal indice *)
   i:=(prim+ult+1)DIV 2;
   IF a[i]=INTEGER(i) THEN RETURN i
   ELSIF a[i]>INTEGER(i) THEN RETURN Localiza(a,prim,i-1)
   ELSE RETURN Localiza(a,i+1,ult)
   END;
END Localiza;
```

Tal método sigue la técnica Divide y Vencerás puesto que en cada invocación reduce el problema a uno más pequeño, buscando la posición pedida en un subvector con la mitad de elementos. Este hecho hace que su complejidad sea de orden logarítmico, puesto que su tiempo de ejecución viene dado por la expresión:

$$T(n) = T(n/2) + A$$

siendo A una constante. Nuestro caso base sigue siendo el mismo, por lo que podemos volver a tomar $T(1) = 1$. Para resolver la ecuación hacemos el cambio $t_k = T(2^k)$, por lo que

$$t_k = t_{k-1} + A,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-1)^2 = 0$. Aplicando los métodos utilizados en el capítulo 1, la expresión de t_k viene dada por

$$t_k = c_1 k + c_2,$$

y de aquí, deshaciendo el cambio $n = 2^k$ (o lo que es igual, $k = \log n$), obtenemos:

$$T(n) = c_1 \log n + c_2 \in O(\log n).$$

3.10 REPETICIÓN DE CÁLCULOS EN FIBONACCI

En el cálculo recursivo del n-ésimo número de Fibonacci, $fib(n)$, necesitamos determinar para cada $0 \le k < n$ el número de veces que se calcula $fib(k)$.

Solución (☺)

Para el cálculo recursivo de $fib(n)$ podemos utilizar la ecuación en recurrencia:

$$fib(n) = fib(n-1) + fib(n-2) \qquad (n > 1)$$

con las condiciones iniciales $fib(1) = fib(0) = 1$. Por tanto, el número de veces que se va a calcular un número $fib(k)$ en el cómputo de $fib(n)$ coincidirá con el número de veces que se calcule en el cómputo de $fib(n-1)$ más el número de veces que se calcule en el cómputo de $fib(n-2)$. En consecuencia, llamando $N_k(n)$ al número de veces que se calcula $fib(k)$ en el cómputo de $fib(n)$, obtenemos la ecuación en recurrencia:

$$N_k(n) = N_k(n-1) + N_k(n-2) \qquad (1 \le k \le n),$$

que es a su vez una ecuación de Fibonacci. Sin embargo sus condiciones iniciales son diferentes, pues para $k = n$ y $k = n-1$ los números $fib(n)$ y $fib(n-1)$ sólo se calculan una vez, con lo cual obtenemos que $N_n(n) = N_{n-1}(n) = 1$ para todo n. Además, $N_k(n) = 0$ si $k > n$. Así, vamos obteniendo:

$N_n(n) = N_{n-1}(n) = 1$	para todo n>1.
$N_{n-2}(n) = N_{n-2}(n-1) + N_{n-2}(n-2) = 1 + 1 = 2$	por la ecuación anterior.
$N_{n-3}(n) = N_{n-3}(n-1) + N_{n-3}(n-2) = 2 + 1 = 3$	por las ecuaciones anteriores.
$N_{n-4}(n) = N_{n-4}(n-1) + N_{n-4}(n-2) = 3 + 2 = 5$	por las ecuaciones anteriores.

De esta forma llegamos a la expresión de $N_k(n)$:

$$N_k(n) = fib(n-k), (1 \le k \le n).$$

Respecto al caso especial $N_0(n)$, su valor se calculará tantas veces como se calcule *fib*(2) en el cómputo de *fib*(*n*), puesto que no hará falta calcularlo en el cómputo de *fib*(1). Por tanto,

$$N_0(n) = N_2(n) = fib(n-2).$$

Es importante señalar en este punto que los resultados obtenidos en este apartado muestran la ineficiencia del algoritmo puramente recursivo para el cálculo de los números de Fibonacci, no sólo por el mero hecho del uso de la pila de ejecución por ser recursivo, sino por la enorme repetición de los cálculos que se realizan. Evitar esta repetición para conseguir tiempos de ejecución polinómicos en vez de exponenciales es una de las ideas claves de la técnica de Programación Dinámica, que será discutida en el capítulo 5.

3.11 EL ELEMENTO MAYORITARIO

Sea $a[1..n]$ un vector de enteros. Un elemento x se denomina *elemento mayoritario* de a si x aparece en el vector más de $n/2$ veces, es decir, $Card\{i \mid a[i]=x\} > n/2$. Necesitamos implementar un algoritmo capaz de decidir si un vector dado contiene un elemento mayoritario (no puede haber más de uno) y calcular su tiempo de ejecución.

Solución (☞)

Al pensar en una posible solución a este ejercicio podemos considerar primero lo que ocurre cuando el vector está ordenado. En ese caso la solución es trivial pues los elementos iguales aparecen juntos. Basta por tanto recorrer el vector buscando un rellano de longitud mayor que $n/2$. Utilizamos el término "rellano" en el mismo sentido que lo hace Gries en su problema *El rellano más largo*: aquel subvector cuyos elementos son todos iguales [GRI81]. El algoritmo puede ser implementado en este caso como sigue:

```
PROCEDURE Mayoritario(VAR a:vector;prim,ult:CARDINAL):BOOLEAN;
(* supone que el vector esta ordenado *)
   VAR mitad,i:CARDINAL;
BEGIN
   IF prim=ult THEN RETURN TRUE END;
   mitad:=(prim+ult+1)DIV 2;
   FOR i:=mitad TO ult DO
      IF a[i]=a[i-mitad+prim] THEN RETURN TRUE END;
   END;
   RETURN FALSE;
END Mayoritario;
```

Este procedimiento comprueba si el vector $a[prim..ult]$ contiene un elemento mayoritario o no, y supone para ello que dicho vector está ordenado en forma creciente.

Por tanto, una primera solución al problema consiste en ordenar el vector y después ejecutar la función anterior. La complejidad de esta solución es de orden

O($n$log$n$), pues de este orden son los procedimientos que ordenan un vector. La complejidad del procedimiento *Mayoritario* no influye frente a ella por ser de orden lineal.

En general, llamaremos algoritmos "en línea" (del término inglés *scanning*) a los algoritmos para vectores que resuelven el problema recorriéndolo una sola vez, sin utilizar otros vectores auxiliares y que permiten en cualquier momento dar una respuesta al problema para la subsecuencia leída hasta entonces. El anterior es un claro ejemplo de este tipo de algoritmos.

Otro algoritmo también muy intuitivo cuando el vector está ordenado es el siguiente: en caso de haber elemento mayoritario, éste ha de encontrarse en la posición $(n+1)\div 2$. Basta entonces con recorrer el vector desde ese elemento hacia atrás y hacia delante, contando el número de veces que se repite. Si este número es mayor que $n/2$, el vector tiene elemento mayoritario. Sin embargo, la complejidad de este algoritmo coincide con la del anterior por tener que ordenar primero el vector, por lo que no representa ninguna mejora.

Sin suponer que el vector se encuentra ordenado, una forma de plantear la solución a este problema aparece indicada en [WEI95]. La idea consiste en encontrar primero un posible candidato, es decir, el único elemento que podría ser mayoritario, y luego comprobar si realmente lo es. De no serlo, el vector no admitiría elemento mayoritario, como le ocurre al vector [1,1,2,2,3,3,3,3,4].

Para encontrar el candidato, suponiendo que el número de elementos del vector es par, vamos a ir comparándolos por pares (a[1] con a[2], a[3] con a[4], etc.). Si para algún k = 1,3,5,7,... se tiene que $a[k] = a[k+1]$ entonces copiaremos $a[k]$ en un segundo vector auxiliar b. Una vez recorrido todo el vector a, buscaremos recursivamente un candidato para el vector b, y así sucesivamente. Este método obecede a la técnica de Divide y Vencerás pues en cada paso el número de elementos se reduce a menos de la mitad.

Vamos a considerar tres cuestiones para implementar un algoritmo basado en esta idea: (*i*) su caso base, (*ii*) lo que ocurre cuando el número de elementos es impar, y (*iii*) cómo eliminar el uso recursivo del vector auxiliar b para no aumentar excesivamente la complejidad espacial del método.

i) En primer lugar, el caso base de la recursión ocurre cuando disponemos de un vector con uno o dos elementos. En este caso existe elemento mayoritario si los elementos del vector son iguales.

ii) Si el número de elementos n del vector es impar y mayor que 2, aplicaremos la idea anterior para el subvector compuesto por sus primeros $n-1$ elementos. Como resultado puede que obtengamos que dicho subvector contiene un candidato a elemento mayoritario, con lo cual éste lo será también para el vector completo. Pero si la búsqueda de candidato para el subvector de $n-1$ elementos no encuentra ninguno, escogeremos como candidato el n-ésimo elemento.

iii) Respecto a cómo eliminar el vector auxiliar b, podemos pensar en utilizar el propio vector a para ir almacenando los elementos que vayan quedando tras cada una de las pasadas.

Esto da lugar al siguiente algoritmo:

```
PROCEDURE Mayoritario2(VAR a:vector;prim,ult:CARDINAL):BOOLEAN;
```

```
(* comprueba si a[prim..ult] contiene un elemento mayoritario *)
   VAR suma,i:CARDINAL; candidato: INTEGER;
BEGIN
   suma:=0;
   IF BuscaCandidato(a,prim,ult,candidato) THEN
      (* comprobacion de si el candidato es o no mayoritario *)
      FOR i:=prim TO ult DO
         IF a[i]=candidato THEN INC(suma) END;
      END
   END;
   RETURN suma>((ult-prim+1)DIV 2);
END Mayoritario2;
```

La función *BuscaCandidato* intenta encontrar un elemento mayoritario:

```
PROCEDURE BuscaCandidato(VAR a:vector;prim,ult:CARDINAL;
                         VAR candidato:INTEGER):BOOLEAN;
   VAR i,j:CARDINAL;
BEGIN
   candidato:=a[prim];
   IF ult<prim THEN RETURN FALSE END;  (* casos base *)
   IF ult=prim THEN RETURN TRUE END;
   IF prim+1=ult THEN
      candidato:=a[ult];
      RETURN (a[prim]=a[ult])
   END;
   j:=prim; (* caso general *)
   IF ((ult-prim+1)MOD 2)=0 THEN (* n par *)
      FOR i:=prim+1 TO ult BY 2 DO
         IF a[i-1]=a[i] THEN
            a[j]:=a[i]; INC(j)
         END
      END;
      RETURN BuscaCandidato(a,prim,j-1,candidato);
   ELSE (* n impar *)
      FOR i:=prim TO ult-1 BY 2 DO
         IF a[i]=a[i+1] THEN a[j]:=a[i]; INC(j) END
      END;

      IF NOT BuscaCandidato(a,prim,j-1,candidato) THEN
         candidato:=a[ult]
      END;
      RETURN TRUE
   END;
END BuscaCandidato;
```

La complejidad del algoritmo *BuscaCandidato* es de orden O(n), pues en cada iteración del procedimiento general se efectúa un ciclo de orden n, junto con una llamada recursiva a la función, pero a lo sumo con n/2 elementos. Esto permite expresar su tiempo de ejecución $T(n)$ mediante la ecuación

$$T(n) = T(n/2) + An + B,$$

siendo A y B constantes. Para resolver la ecuación hacemos el cambio $t_k = T(2^k)$, por lo que

$$t_k = t_{k-1} + A2^k + B,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-1)^2(x-2)=0$. Aplicando los métodos utilizados en el capítulo 1, la expresión de t_k viene dada por

$$t_k = c_1k + c_2 + c_3 2^k,$$

y de aquí, deshaciendo el cambio $n = 2^k$ (o lo que es igual, $k = \log n$), obtenemos:

$$T(n) = c_1\log n + c_2 + c_3 n \in O(n).$$

Este algoritmo es, por tanto, mejor que el que ordena primero el vector.

3.12 LA MODA DE UN VECTOR

Deseamos implementar un algoritmo Divide y Vencerás para encontrar la *moda* de un vector, es decir, aquel elemento que se repite más veces.

Solución

La primera solución que puede plantearse para resolver este problema es a partir de la propia definición de moda. Se calcula la frecuencia con la que aparece cada uno de los elementos del vector y se escoge aquel que se repite más veces. Esto da lugar a la siguiente función:

```
PROCEDURE Moda(VAR a:vector;prim,ult:CARDINAL):INTEGER;
   VAR i,frec,maxfrec:CARDINAL;moda:INTEGER;
BEGIN
   IF prim=ult THEN RETURN a[prim] END;
   moda:=a[prim];
   maxfrec:=Frecuencia(a,a[prim],prim,ult);

   FOR i:=prim+1 TO ult DO
      frec:=Frecuencia(a,a[i],i,ult);
      IF frec>maxfrec THEN
         maxfrec:=frec;
         moda:=a[i]
      END;
   END;
```

```
   RETURN moda
END Moda;
```

La función *Frecuencia* es la que calcula el número de veces que se repite un elemento dado:

```
PROCEDURE Frecuencia(VAR
a:vector;p:INTEGER;prim,ult:CARDINAL):CARDINAL;
   VAR i,suma:CARDINAL;
BEGIN
   IF prim>ult THEN RETURN 0 END;
   suma:=0;
   FOR i:=prim TO ult DO
      IF a[i]=p THEN
         INC(suma)
      END;
   END;
   RETURN suma;
END Frecuencia;
```

La complejidad de la función *Frecuencia* es O(n), lo que hace que la complejidad del algoritmo presentado para calcular la moda sea de orden O(n^2).

Ahora bien, en el caso particular en el que el vector esté ordenado existe una forma mucho más eficiente para calcular su moda, recorriendo el vector una sola vez. El algoritmo que presentamos a continuación es del tipo "en línea" y está basado en el algoritmo para calcular el rellano más largo de un vector (ver [GRI81] y el problema anterior), y da lugar a la siguiente función:

```
PROCEDURE Moda2(VAR a:vector;prim,ult:CARDINAL):INTEGER;
(* supone que el vector a[prim..ult] esta ordenado *)
   VAR i,p:CARDINAL;moda:INTEGER;
BEGIN
   i:=prim+1; p:=1; moda:=a[prim];
   WHILE i<=ult DO
      IF a[i-p]=a[i] THEN INC(p); moda:=a[i] END;
      INC(i);
   END;
   RETURN moda
END Moda2;
```

La complejidad de este algoritmo es O(n). Sin embargo, como es preciso ordenar primero el vector antes de invocar a esta función, la complejidad del algoritmo resultante sería de orden O($n\log n$).

Existe sin embargo una solución aplicando la técnica de Divide y Vencerás, indicada en [GON91] capaz de ofrecer una complejidad mejor que O($n\log n$).

El algoritmo se basa en la utilización de dos conjuntos, *homog* y *heterog*, que van a contener en cada paso subvectores del vector original *a*[*prim..ult*]. El primero

de ellos contiene sólo aquellos subvectores que tienen todos sus elementos iguales, y el segundo aquellos que tienen sus elementos distintos.

Inicialmente *homog* es vacío y *heterog* contiene al vector completo. En cada paso vamos a extraer el subvector p de *heterog* de mayor longitud, calcularemos su mediana y lo dividiremos en tres subvectores: $p1$, que contiene los elementos de p menores a su mediana, $p2$, con los elementos de p iguales a su mediana, y $p3$, con los elementos de p mayores a su mediana. Entonces actualizaremos los conjuntos *homog* y *heterog*, pues en el primero introduciremos $p2$ y en el segundo $p1$ y $p3$.

Este proceso lo repetiremos mientras que la longitud del subvector más largo de *heterog* sea mayor que la del más largo de *homog*. Una vez llegado a este punto, el subvector más largo de *homog* contendrá la moda del vector original.

Para implementar tal esquema haremos uso de un tipo abstracto de datos que representa a los conjuntos de subvectores (*CJTS*), que aporta las operaciones sobre los elementos de tal tipo, que supondremos implementado. Los subvectores van a ser representados como ternas en donde el primer elemento es un vector, y los otros dos indican las posiciones de comienzo y fin de sus elementos.

El algoritmo que desarrolla esta idea para encontrar la moda de un vector es el siguiente:

```
PROCEDURE Moda3(VAR a:vector;prim,ult:CARDINAL):INTEGER;
   VAR  p,p1,p2,p3:CJTS.subvector;
        homog,heterog:CJTS.conjunto;
        mediana:INTEGER;
        izq,der:CARDINAL;
BEGIN
   CJTS.Crear(homog);
   CJTS.Crear(heterog);
   (* insertamos a[prim..ult] en heterog: *)
   p.a:=a;
   p.prim:=prim;
   p.ult:=ult;
   CJTS.Insertar(heterog,p);
   WHILE CJTS.Long_Mayor(heterog)> CJTS.Long_Mayor(homog) DO
      p:=CJTS.Mayor(heterog); (* esto extrae p del conjunto *)
      (* calculamos la mediana de p *)
      mediana:=Kesimo(p.a,p.prim,p.ult,(p.ult-p.prim+2)DIV 2);
      (* y dividimos p en 3 subvectores *)
      Pivote2(p.a,mediana,p.prim,p.ult,izq,der);
      p1.a:=p.a; p1.prim:=p.prim; p1.ult:=izq-1;
      p2.a:=p.a; p2.prim:=izq; p2.ult:=der-1;
      p3.a:=p.a; p3.prim:=der; p3.ult:=p.ult;
      (* ahora modificamos los conjuntos heterog y homog *)
      IF p1.prim<p1.ult THEN CJTS.Insertar(heterog,p1) END;
      IF p3.prim<p3.ult THEN CJTS.Insertar(heterog,p3) END;
      IF p2.prim<p2.ult THEN CJTS.Insertar(homog,p2) END
   END; (* WHILE *)
   IF CJTS.Esvacio(homog) THEN
```

```
      RETURN a[prim]
    END;
    p:=CJTS.Mayor(homog);
    CJTS.Destruir(homog);
    CJTS.Destruir(heterog);
    RETURN p.a[p.prim]
  END Moda3;
```

Las funciones *Kesimo* y *Pivote2* fueron definidas e implementadas en el capítulo anterior, y son utilizadas aquí para calcular la mediana del vector y dividirlo en tres partes, de acuerdo al esquema general presentado.

El estudio de la complejidad de este algoritmo no es fácil. Sólo mencionaremos que su complejidad, como se muestra en [GON91], es de orden $O(n\log(n/m))$, siendo m la multiplicidad de la moda, y que por las constantes multiplicativas que posee, resulta ser mejor que el algoritmo *Moda2* presentado anteriormente. Sin embargo, la dificultad de su diseño e implementación han de tenerse también en cuenta, pues complican notablemente su codificación y mantenimiento.

3.13 EL TORNEO DE TENIS

Necesitamos organizar un torneo de tenis con n jugadores en donde cada jugador ha de jugar exactamente una vez contra cada uno de sus posibles n–1 competidores, y además ha de jugar un partido cada día, teniendo a lo sumo un día de descanso en todo el torneo. Por ejemplo, las siguientes tablas son posibles cuadrantes resultado para torneos con 5 y 6 jugadores:

Jug	d1	d2	d3	d4	d5
1	2	3	4	5	–
2	1	5	3	–	4
3	–	1	2	4	5
4	5	–	1	3	2
5	4	2	–	1	3

Jug	d1	d2	d3	d4	d5
1	2	3	4	5	6
2	1	5	3	6	4
3	6	1	2	4	5
4	5	6	1	3	2
5	4	2	6	1	3
6	3	4	5	2	1

Solución (😐)

Para resolver este problema procederemos por partes, considerando los siguientes casos:

a) Si n es potencia de 2, implementaremos un algoritmo para construir un cuadrante de partidas del torneo que permita terminarlo en n–1 días.

b) Dado cualquier $n>1$, implementaremos un algoritmo para construir un cuadrante de partidas del torneo que permita terminarlo en $n-1$ días si n es par, o en n días si n es impar.

En el primer caso suponemos que n es una potencia de 2. El caso más simple se produce cuando sólo tenemos dos jugadores, cuya solución es fácil pues basta enfrentar uno contra el otro.

Si $n>2$, aplicaremos la técnica de Divide y Vencerás para construir la tabla pedida suponiendo que tenemos calculada ya una solución para la mitad de los jugadores, esto es, que tenemos relleno el cuadrante superior izquierdo de la tabla. En este caso los otros tres cuadrantes no son difíciles de rellenar, como puede observarse en la siguiente figura, y en donde se han tenido en cuenta la siguientes consideraciones para su construcción:

1. El cuadrante inferior izquierdo debe enfrentar a los jugadores de número superior entre ellos, por lo que se obtiene sumando $n/2$ a los valores del cuadrante superior izquierdo.
2. El cuadrante superior derecho enfrenta a los jugadores con menores y mayores números, y se puede obtener enfrentando a los jugadores numerados 1 a $n/2$ contra $(n/2)+1$ a n respectivamente en el día $n/2$, y después rotando los valores $(n/2)+1$ a n cada día.
3. Análogamente, el cuadrante inferior derecho enfrenta a los jugadores de mayor número contra los de menor número, y se puede obtener enfrentando a los jugadores $(n/2)+1$ a n contra 1 a $n/2$ respectivamente en el día $n/2$, y después rotando los valores 1 a n cada día, pero en sentido contrario a como lo hemos hecho para el cuadrante superior derecho.

	d1
J1	2
J2	1

	d1	d2	d3
J1	2	3	4
J2	1	4	3
J3	4	1	2
J4	3	2	1

	d1	d2	d3	d4	d5	d6	d7
J1	2	3	4	5	6	7	8
J2	1	4	3	6	7	8	5
J3	4	1	2	7	8	5	6
J4	3	2	1	8	5	6	7
J5	6	7	8	1	4	3	2
J6	5	8	7	2	1	4	3
J7	8	5	6	3	2	1	4
J8	7	6	5	4	3	2	1

El algoritmo que implementa tal estrategia es:

```
CONST MAXJUG =...; (* numero maximo de jugadores *)
TYPE cuadrante = ARRAY [1..MAXJUG],[1..MAXJUG] OF CARDINAL;

PROCEDURE Torneo(n:CARDINAL;VAR tabla:cuadrante);
(* n es el numero de jugadores, que suponemos potencia de 2 *)
(* en tabla devuelve el cuadrante de partidos relleno *)
   VAR jug,dia:CARDINAL;
BEGIN
   IF n=2 THEN   (* caso base *)
```

```
      tabla[1,1]:=2;
      tabla[2,1]:=1
   ELSE
      (* primero se rellena el cuadrante superior izquierdo *)
      Torneo(n DIV 2, tabla); (* llamada recursiva *)
      (* despues el cuadrante inferior izquierdo *)
      FOR jug:=(n DIV 2)+1 TO n DO
         FOR dia:=1 TO (n DIV 2)-1 DO
            tabla[jug,dia]:=tabla[jug-(n DIV 2),dia]+(n DIV 2)
         END
      END;
      (* luego el cuadrante superior derecho *)
      FOR jug:=1 TO (n DIV 2) DO
         FOR dia:=(n DIV 2) TO n-1 DO
            IF (jug+dia)<=n THEN tabla[jug,dia]:=jug+dia
            ELSE tabla[jug,dia]:=jug+dia-(n DIV 2)
            END
         END
      END;
      (* y finalmente el cuadrante inferior derecho *)
      FOR jug:=(n DIV 2)+1 TO n DO
         FOR dia:=(n DIV 2) TO n-1 DO
            IF jug>dia THEN tabla[jug,dia]:=jug-dia
            ELSE tabla[jug,dia]:=(jug+(n DIV 2))-dia
            END
         END
      END
   END (* IF *)
END Torneo;
```

Supongamos ahora que el número de jugadores n es impar y que sabemos resolver el problema para un número par de jugadores. En este caso existe una solución al problema en n días, que se construye a partir de la solución al problema para n+1 jugadores. Si n es impar entonces n+1 es par, y sea $S[1..n+1][1..n]$ el cuadrante solución para n+1 jugadores. Entonces podemos obtener el cuadrante solución para n jugadores $T[1..n][1..n]$ como:

$$T[jug, dia] = \begin{cases} S[jug, dia] & \text{si } S[jug, dia] \neq n+1 \\ 0 & \text{si } S[jug, dia] = n+1 \end{cases}$$

Es decir, utilizamos la convención de que un 0 en la posición $[i,j]$ de la tabla indica que el jugador i descansa (no se enfrenta a nadie) el día j, y aprovechamos este hecho para construir el cuadrante solución pedido. Por ejemplo, para el caso de $n = 3$ nos apoyamos en la tabla construida para 4 jugadores, eliminando la última fila y sustituyendo las apariciones del jugador número 4 por ceros:

	d1	d2	d3
J1	2	3	0
J2	1	0	3
J3	0	1	2

Sólo nos queda resolver el caso en que *n* es par, y para ello llamaremos *m* al número *n*÷2. Utilizando la técnica de Divide y Vencerás vamos a encontrar una forma de resolver el problema para *n* jugadores suponiendo que lo tenemos resuelto para *m*. Distinguiremos dos casos:

Si *m* es par, sabemos que existe una solución para enfrentar a esos *m* jugadores entre sí en *m*–1 días. Éste va a constituir el cuadrante superior izquierdo de la solución para *n*. Los otros tres cuadrantes se van a construir de igual forma al caso anterior cuando *n* es potencia de 2.

Si *m* es impar, su solución necesita *m* días. Esto va a volver a constituir el cuadrante superior izquierdo de la solución para el caso de *n* jugadores, pero ahora nos encontramos con que tiene algunos ceros, que necesitaremos rellenar convenientemente.

El cuadrante inferior izquierdo va a construirse como anteriormente, es decir, va a enfrentar a los jugadores de número superior entre ellos, por lo que se obtiene sumando *n*/2 a los valores del cuadrante superior que no sean cero.

El cuadrante superior derecho enfrenta a los jugadores con menores y mayores números y se va a obtener de forma similar a como lo construíamos antes, solo que no va a enfrentar a los jugadores 1 a *n*/2 contra (*n*/2)+1 a *n* en el día *n*/2, sino en cada una de las posiciones vacías del cuadrante superior izquierda. Los demás días del cuadrante superior derecho sí se van a obtener rotando esos valores cada día.

Análogamente, el cuadrante inferior derecho enfrenta a los jugadores de mayor número contra los de menor número, y se va a obtener enfrentando a los jugadores (*n*/2)+1 a *n* contra 1 a *n*/2 respectivamente, pero no en el día *n*/2, sino ocupando las posiciones vacías del cuadrante inferior izquierdo. Los valores restantes sí se obtendrán como antes, rotando los valores 1 a *n* cada día, pero en sentido contrario a como lo hemos hecho para el cuadrante superior.

Este proceso puede apreciarse en la siguiente figura para *n* = 6:

	d1	d2	d3
J1	2	3	0
J2	1	0	3
J3	0	1	2

m = 3

	d1	d2	d3	d4	d5
J1	2	3	0		
J2	1	0	3		
J3	0	1	2		
J4	5	6	0		
J5	4	0	6		
J6	0	4	5		

cuadrantes 1º y 2º.

	d1	d2	d3	d4	d5
J1	2	3	4	5	6
J2	1	5	3	6	4
J3	6	1	2	4	5
J4	5	6	1	3	2
J5	4	2	6	1	3
J6	3	4	5	2	1

cuadrantes 3º y 4º.

y el algoritmo que lleva a cabo tal estrategia puede ser implementado como sigue:

```
PROCEDURE Torneo(n:CARDINAL; VAR tabla:cuadrante);
(* n es el num. jugadores y en tabla se devuelve la solucion *)
   VAR jug,dia,m:CARDINAL;
BEGIN
```

```
IF n=2 THEN   (* caso base *)
   tabla[1,1]:=2; tabla[2,1]:=1
ELSIF (n MOD 2)<>0 THEN  (* n impar *)
   Torneo(n+1,tabla);    (* llamada recursiva *)
   FOR jug:=1 TO n DO    (* eliminamos el jugador n+1 *)
      FOR dia:=1 TO n DO
         IF tabla[jug,dia]=n+1 THEN tabla[jug,dia]:=0 END
      END
   END
ELSE (* n par *)
   m:=n DIV 2;
   Torneo(m, tabla); (* primero el cuadrante sup. izq. *)
   IF (m MOD 2 )=0 THEN  (* m par *)
      FOR jug:=m+1 TO n DO  (* cuadrante inferior izquierdo *)
         FOR dia:=1 TO m-1 DO
            tabla[jug,dia]:=tabla[jug-m,dia]+m
         END
      END;
      FOR jug:=1 TO m DO (* cuadrante superior derecho *)
         FOR dia:=m TO n-1 DO
            IF (jug+dia)<=n THEN tabla[jug,dia]:=jug+dia
            ELSE tabla[jug,dia]:=jug+dia-m
            END
         END
      END;
      FOR jug:=m+1 TO n DO  (* y cuadrante inferior derecho *)
         FOR dia:=m TO n-1 DO
            IF jug>dia THEN tabla[jug,dia]:=jug-dia
            ELSE tabla[jug,dia]:=(jug+m)-dia
            END
         END
      END
   ELSE (* m impar *)
      FOR jug:=m+1 TO n DO  (* cuadrante inferior izquierdo *)
         FOR dia:=1 TO m DO
            IF tabla[jug-m,dia]=0 THEN tabla[jug,dia]:=0
            ELSE tabla[jug,dia]:=tabla[jug-m,dia]+m
            END
         END
      END;
      FOR jug:=1 TO m DO (* ceros de los cuadrantes izq *)
         FOR dia:=1 TO m DO
            IF tabla[jug,dia]=0 THEN
               tabla[jug,dia]:=jug+m;
               tabla[jug+m,dia]:=jug
            END
```

```
          END
        END;
        FOR jug:=1 TO m DO (* cuadrante superior derecho *)
          FOR dia:=m+1 TO n-1 DO
            IF (jug+dia)<=n THEN tabla[jug,dia]:=jug+dia
            ELSE tabla[jug,dia]:=jug+dia-m
            END
          END
        END;
        FOR jug:=m+1 TO n DO  (* ultimo, cuadrante inf. der. *)
          FOR dia:=m+1 TO n-1 DO
            IF jug>dia THEN tabla[jug,dia]:=jug-dia
            ELSE tabla[jug,dia]:=(jug+m)-dia
            END
          END
        END
      END (* IF m impar *)
    END (* IF n par *)
  END Torneo;
```

Este algoritmo puede reducirse en extensión pero a costa de perder claridad en los casos tratados y en el manejo de los índices que rellenan la tabla solución. Hemos preferido mantener la presente versión para una mejor legibilidad del algoritmo.

Por otro lado, este es un ejemplo de algoritmo Divide y Vencerás de simplificación, esto es, en donde el problema se reduce en cada paso a un solo problema de tamaño más pequeño, cuyo proceso de combinación no es trivial.

3.14 DIVIDE Y VENCERÁS MULTIDIMENSIONAL

Una generalización de la técnica que estamos estudiando en este capítulo es el Divide y Vencerás multidimensional, la cual trata de resolver un problema de tamaño n en un espacio k-dimensional mediante la solución de dos subproblemas de tamaño $n/2$ en un espacio k-dimensional y un problema de tamaño n en un espacio $(k-1)$-dimensional. Para ilustrar esta técnica vamos a considerar el siguiente problema:

En un espacio discreto bidimensional [1..M]x[1..M] tenemos un conjunto S de n puntos. Diremos que un punto $P=(p_1,p_2)$ *domina* a otro punto $Q=(q_1,q_2)$ si $p_1>q_1$ y $p_2>q_2$. El *rango* de un punto P de S es el número de puntos que domina. Implementar un algoritmo que calcule el rango de todos los puntos del conjunto S.

Solución (☞)

Una primera solución al problema consiste en calcular el rango para cada punto comparándolo con los $n-1$ restantes, y da lugar al siguiente algoritmo:

```
CONST M =...; (* dimension del espacio *)
```

```
      n =...; (* numero de puntos *)
TYPE  punto = RECORD x,y:[1..M] END;
      cjt_puntos = ARRAY[1..n] OF punto;
      rango = ARRAY[1..n] OF CARDINAL;

PROCEDURE CalculaRango(s:cjt_puntos;prim,ult:CARDINAL;VAR r:rango);
(* calcula en r el rango de los puntos en el conj. s[prim..ult] *)
   VAR i,j:CARDINAL;
BEGIN
   FOR i:=prim TO ult DO
      r[i]:=0;
      FOR j:=prim TO ult DO
         IF Domina(s[i],s[j]) THEN INC(r[i]) END;
      END;
   END;
END CalculaRango;
```

Este procedimiento usa una función que decide cuándo un punto domina a otro:

```
PROCEDURE Domina(p,q:punto):BOOLEAN;
BEGIN
   RETURN (p.x>q.x) AND (p.y>q.y)
END Domina;
```

La complejidad de la función *CalculaRango* es claramente de orden $O(n^2)$ por tratarse de dos bucles anidados en donde sólo se realizan operaciones de orden constante. Sin embargo, existe un método de menor complejidad utilizando Divide y Vencerás multidimensional, originalmente expuesto en [BEN80].

En primer lugar se escoge una línea vertical *L* que divide al conjunto de puntos *S* en dos subconjuntos A y B, cada uno con la mitad de los puntos (la ecuación de esta recta no es sino $x = med$, siendo *med* la mediana del conjunto de abcisas de los puntos de *S*).

El segundo paso del método calcula recursivamente el rango de los dos subconjuntos.

Por último, el tercer paso combina los resultados obtenidos para componer la solución del problema. Para esto hemos de fijarnos en dos hechos:

a) Primero, que ningún punto de A domina a uno de B (pues la abcisa de un punto de A nunca es mayor que la de cualquier punto de B).
b) Segundo, que un punto *P* de B va a dominar a otro *Q* de A si y sólo si la ordenada de *P* es mayor que la de *Q*.

Por el primero de ellos, el rango de los puntos de A coincide con el rango que van a tener cuando los consideremos puntos de *S*. Ahora bien, para calcular el rango final de los puntos de B necesitamos añadir al rango calculado para cada uno de ellos el número de puntos de A que cada uno domina. Para encontrar tal número lo que haremos es "proyectar" los puntos de *S* sobre la línea *L*. Una vez hecho esto,

basta recorrer tal línea de abajo arriba e ir acumulando el número de puntos de A que vayamos encontrando. Cuando se encuentre un punto de B, el número de puntos de A acumulado hasta ese momento será el número pedido.

Obsérvese cómo este método sigue la estrategia de Divide y Vencerás multidimensional. Para resolver un problema de tamano n en el plano resolvemos dos problemas de tamaño $n/2$ en el plano, y uno de tamaño n sobre una recta. Para el caso de la recta (dimensión 1), el cálculo del rango de cada uno de los puntos es fácil: basta con ordenarlos y el rango de un punto va a ser el número de puntos que le preceden.

Para implementar este algoritmo vamos a hacer dos suposiciones que no van a restar ninguna generalidad a la solución, pero que permiten simplificar el código. Lo primero que supondremos es que las abcisas de los puntos son todas distintas, y que están numeradas consecutivamente de 1 a n. La segunda es que el conjunto S está ordenado por los valores de las abcisas de sus puntos. Ninguna de ellas resta generalidad. La primera, porque podemos utilizar claves distintas para referenciar las abcisas de los puntos. Respecto a la segunda, podemos ordenar el conjunto S antes de invocar a este algoritmo, lo que únicamente conlleva una complejidad adicional de orden $O(n\log n)$.

Tales suposiciones nos van a permitir encontrar la línea L fácilmente (la mediana es el elemento en posición $(n+1)\div 2$ del vector), y recorrerla posteriormente de forma creciente sin problemas. Este esquema da lugar al siguiente procedimiento:

```
PROCEDURE CalculaRango2(s:cjt_puntos;prim,ult:CARDINAL;
                        VAR r:rango);
(* calcula en r el rango de los puntos en s[prim..ult] *)
(* supone que los puntos estan ordenados respecto a sus abcisas,
   y que estas coinciden con prim,prim+1,...,ult. *)
   VAR i,j,mitad,suma_A:CARDINAL;
       s_y:cjt_puntos;
BEGIN
   IF prim=ult THEN (* caso base *)
      r[prim]:=0; RETURN
   END;
(* paso 2: resolver el problema para los subconjuntos A y B *)
   mitad:=(prim+ult) DIV 2;
   CalculaRango2(s,prim,mitad,r);
   CalculaRango2(s,mitad+1,ult,r);
   (* paso 3: ordenamos s respecto a su ordenada *)
   s_y:=s; (* utilizamos una copia de s para esto *)
   Ordenar_Y(s_y,prim,ult);
   (* paso 4: y ahora recorremos la linea imaginaria L *)
   suma_A:=0;
   FOR i:=prim TO ult DO
      IF s_y[i].x<=mitad THEN INC(suma_A) (* el punto es de A *)
      ELSE INC(r[s_y[i].x],suma_A);       (* el punto es de B *)
      END;
```

```
    END;
  END CalculaRango2;
```

El procedimiento *Ordenar_Y(VAR a:cjt_puntos; prim,ult:CARDINAL)* ordena el conjunto de puntos *a*[*prim..ult*] respecto a su ordenada.

Para analizar el tiempo de ejecución $T(n)$ del procedimiento *CalculaRango2*, calcularemos la complejidad de cada uno de sus pasos:

- El caso base se resuelve con 4 operaciones elementales; es por tanto O(1).
- El paso 2 tiene un tiempo de ejecución $2T(n/2)+O(1)$.
- El paso 3 conlleva una copia del vector y una ordenación, es decir: $O(n)+O(n\log n)$
- Por último, el orden de complejidad del paso 4 es $O(n)$.

Por consiguiente, el tiempo de ejecución viene dado por la ecuación en recurrencia

$$T(n) = O(1)+2T(n/2)+O(1)+O(n)+O(n\log n)+O(n) = 2T(n/2)+O(n\log n),$$

cuya solución es de un orden de complejidad $O(n\log^2 n)$. Para ver esto, podemos expresar $T(n)$ como:

$$T(n) = 2T(n/2) + An\log n,$$

siendo A una constante. Llamando $n = 2^k$ (o lo que es igual, $k = \log n$) y haciendo el cambio $t_k = T(2^k)$, obtenemos

$$t_k = 2t_{k-1} + Ak2^k$$

ecuación en recurrencia no homogénea cuya ecuación característica es $(x-2)^3 = 0$, lo que implica que t_k viene dado por la expresión:

$$t_k = c_1 2^k + c_2\, k2^k + c_3 k^2 2^k.$$

Deshaciendo los cambios realizados con anterioridad obtenemos finalmente:

$$T(n) = c_1 n + c_2 n\log n + c_3 n\log^2 n \in O(n\log^2 n).$$

Este algoritmo mejora notablemente el presentado al principio de este apartado. Sin embargo, tras un estudio de *CalculaRango2* podemos observar que su complejidad esta dominada por la complejidad de la ordenación que se realiza en su tercer paso. ¿Existe alguna forma de evitar esta ordenación?

La respuesta es afirmativa, y además da lugar a una mejora usual de este tipo de algoritmos. Se basa en ordenar sólo una vez (al principio) el conjunto *S* respecto a las ordenadas de sus puntos, y mantener esta ordenación cuando se divida el conjunto inicial en dos. Esto permite simplificar el paso 3, lo que hace que el tiempo de ejecución del algoritmo sea ahora de $T(n) = 2T(n/2) + O(n)$. Esta ecuación es de una complejidad $O(n\log n)$, como hemos calculado en varios de los problemas de este capítulo.

Cara a implementar esta solución, lo que vamos a necesitar es una estructura auxiliar que nos permita decidir en cualquier momento a qué conjunto (A o B) pertenece un punto de *S*. Esto nos lleva al siguiente algoritmo:

```
PROCEDURE CalculaRango3(sX,sY:cjt_puntos;prim,ult:CARDINAL;
                        VAR r:rango);
(* calcula el rango de los puntos en el conjunto sX[prim..ult] *)
(* supone que los puntos de sX estan ordenados respecto a sus
   abcisas, que estas coinciden con prim,prim+1,...,ult, y que los
   puntos de sY estan ordenados respecto a sus ordenadas *)

   VAR i,j,mitad,suma_A:CARDINAL;s:cjt_puntos;
BEGIN
   IF prim=ult THEN (* caso base *)
      r[prim]:=0; RETURN
   END;
   (* paso 2: resolvemos el problema para los subconjuntos A y B *)
   mitad:=(prim+ult) DIV 2;
   CalculaRango3(sX,sY,prim,mitad,r);
   CalculaRango3(sX,sY,mitad+1,ult,r);
   (* en el paso 3 seleccionamos los elementos adecuados de sY *)
   i:=1;j:=prim;
   WHILE (j<=ult) DO
      IF (prim<=sY[i].x) AND (sY[i].x<=ult) THEN
         s[j]:=sY[i]; INC(j)
      END;
      INC(i)
   END;
   (* paso 4: y ahora recorremos la linea imaginaria L *)
   suma_A:=0;
   FOR i:=prim TO ult DO
      IF s[i].x<=mitad THEN INC(suma_A)     (* el punto es de A *)
      ELSE INC(r[s[i].x],suma_A)           (* el punto es de B *)
      END;
   END;
END CalculaRango3;
```

3.15 LA SUBSECUENCIA DE SUMA MÁXIMA

Dados n enteros cualesquiera $a_1,a_2,...,a_n$, necesitamos encontrar el valor de la expresión:

$$\max_{1 \le i \le j \le n} \left\{ \sum_{k=i}^{j} a_k \right\},$$

que calcula el máximo de las sumas parciales de elementos consecutivos. Como ejemplo, dados los 6 enteros (–2,11,–4,13,–5,–2) la solución al problema es 20 (suma de a_2 hasta a_4).

Deseamos implementar un algoritmo Divide y Vencerás de complejidad $n\log n$ que resuelva el problema. ¿Existe algún otro algoritmo que lo resuelva en menor tiempo?

Solución (☞)

Existe una solución trivial a este problema, basada en calcular todas las posibles sumas y escoger la de valor máximo (esto es, mediante un algoritmo de "fuerza bruta") cuyo orden de complejidad es $O(n^3)$. Esto lo hace bastante ineficiente para valores grandes de n:

```
PROCEDURE Sumamax(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
   VAR izq,der,i:CARDINAL; max_aux,suma:INTEGER;
BEGIN
   max_aux:=0;
   FOR izq:=prim TO ult DO
      FOR der:=izq TO ult DO
         suma:=0;
         FOR i:=izq TO der DO
            suma:=suma+a[i]
         END;
         IF suma>max_aux THEN
            max_aux:=suma
         END
      END
   END;
   RETURN max_aux
END Sumamax;
```

Una mejora inmediata para el algoritmo es la de evitar calcular la suma para cada posible subsecuencia, aprovechando el valor ya calculado de la suma de los valores de $a[izq..der-1]$ para calcular la suma de los valores de $a[izq..der]$. Esto da lugar a la siguiente función

```
PROCEDURE Sumamax2(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
   VAR izq,der:CARDINAL; max_aux,suma:INTEGER;
BEGIN
   max_aux:=0;
   FOR izq:=prim TO ult DO
      suma:=0;
      FOR der:=izq TO ult DO
         suma:=suma+a[der];
```

```
        (* suma contiene la suma de a[izq..der] *)
        IF suma>max_aux THEN max_aux:=suma END
      END
    END;
    RETURN max_aux
  END Sumamax2;
```

cuya complejidad es de orden O(n^2), lo cual mejora sustancialmente al anterior, pero que aún no consigue la complejidad pedida.

Utilizaremos ahora la técnica de Divide y Vencerás para intentar mejorar la eficiencia de los algoritmos anteriores, y lo haremos siguiendo una idea de [BEN89]. Para ello, dividiremos el problema en tres subproblemas más pequeños, sobre cuyas soluciones construiremos la solución total.

En este caso la subsecuencia de suma máxima puede encontrarse en uno de tres lugares. O está en la primera mitad del vector, o en la segunda, o bien contiene al punto medio del vector y se encuentra en ambas mitades. Las tres soluciones se combinan mediante el cálculo de su máximo para obtener la suma pedida.

Los dos primeros casos pueden resolverse recursivamente. Respecto al tercero, podemos calcular la subsecuencia de suma máxima de la primera mitad que contenga al último elemento de esa primera mitad, y la subsecuencia de suma máxima de la segunda mitad que contenga al primer elemento de esa segunda mitad. Estas dos secuencias pueden concatenarse para construir la subsecuencia de suma máxima que contiene al elemento central de vector. Esto da lugar al siguiente algoritmo:

```
PROCEDURE Sumamax3(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
  VAR mitad,i:CARDINAL;
     max_izq,max_der,suma,max_aux:INTEGER;
BEGIN
  (* casos base *)
  IF prim>ult THEN RETURN 0 END;
  IF prim=ult THEN RETURN Max2(0,a[prim]) END;
  mitad:=(prim+ult)DIV 2;
  (* casos 1 y 2 *)
  max_aux:=Max2(Sumamax3(a,prim,mitad),Sumamax3(a,mitad+1,ult));
  (* caso 3: parte izquierda *)
  max_izq:=0;
  suma:=0;
  FOR i:=mitad TO prim BY -1 DO
    suma:=suma+a[i];
    max_izq:=Max2(max_izq,suma)
  END;
  (* caso 3: parte derecha *)
  max_der:=0;
  suma:=0;
  FOR i:=mitad+1 TO ult DO
```

```
      suma:=suma+a[i];
      max_der:=Max2(max_der,suma)
   END;
   (* combinacion de resultados *)
   RETURN Max2(max_der+max_izq,max_aux)
END Sumamax3;
```

donde la función *Max2* utilizada es la que calcula el máximo de dos números enteros.

El procedimiento *Sumamax3* es de complejidad O($n\log n$), puesto que su tiempo de ejecución $T(n)$ viene dado por la ecuación en recurrencia

$$T(n) = 2T(n/2) + An$$

con la condición inicial $T(1) = 7$, siendo A una constante.

Obsérvese además que este análisis es válido puesto que hemos añadido la palabra *VAR* al vector *a* en la definición del procedimiento. Si no, se producirían copias de *a* en las invocaciones recursivas, lo que incrementaría el tiempo de ejecución del procedimiento.

Respecto a la última parte del problema, necesitamos encontrar un algoritmo aún mejor que éste. La clave va a consistir en una modificación a la idea básica del algoritmo anterior, basada en un algoritmo del tipo "en línea" (véase el problema del elemento mayoritario, en este capítulo).

Supongamos que ya tenemos la solución del problema para el subvector $a[prim..i–1]$. ¿Cómo podemos extender esa solución para encontrar la solución de $a[prim..i]$? De forma análoga al razonamiento que hicimos para el algoritmo anterior, la subsecuencia de suma máxima de $a[prim..i]$ puede encontrarse en $a[prim..i–1]$, o bien contener al elemento $a[i]$.

Esto da lugar a la siguiente función:

```
PROCEDURE Sumamax4(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
   VAR i:CARDINAL;
       suma,max_anterior,max_aux:INTEGER;
BEGIN
   max_anterior:=0;
   max_aux:=0;

   FOR i:=prim TO ult DO
      max_aux:=Max2(max_aux+a[i],0);
      max_anterior:=Max2(max_anterior,max_aux)
   END;
   RETURN max_anterior;
END Sumamax4;
```

Este algoritmo no es intuitivo ni fácil de entender a primera vista. La clave del algoritmo se encuentra en la variable *max_aux*, que representa el valor de la suma de la subsecuencia de suma máxima del subvector *a*[*prim..i*] que contiene al elemento *a*[*i*], y que se calcula a partir de su valor para el subvector *a*[*prim..i*–1]. Este valor se incrementa en *a*[*i*] mientras que esa suma sea positiva, pero vuelve a valer cero cada vez que se hace negativa, indicando que la subsecuencia de suma máxima que contiene al elemento *a*[*i*] es la subsecuencia vacía.

El algoritmo resultante es de complejidad lineal, y un análisis detallado de esta función puede encontrase en [GRI80].